The Linux Kernel as a Case Study in Software Evolution

Ayelet Israeli Dror G. Feitelson

Department of Computer Science
The Hebrew University, 91904 Jerusalem, Israel

Abstract

We use 810 versions of the Linux kernel, released over a period of 14 years, to characterize the system's evolution, using Lehman's laws of software evolution as a basis. We investigate different possible interpretations of these laws, as reflected by different metrics that can be used to quantify them. For example, system growth has traditionally been quantified using lines of code or number of functions, but functional growth of an operating system like Linux can also be quantified using the number of system calls. In addition we use the availability of the source code to track metrics, such as McCabe's cyclomatic complexity, that have not been tracked across so many versions previously. We find that the data supports several of Lehman's laws, mainly those concerned with growth and with the stability of the process. We also make some novel observations, e.g. that the average complexity of functions is decreasing with time, but this is mainly due to the addition of many small functions.

Key words: Software evolution, Lehman's laws, Linux kernel

1. Introduction

Software engineering textbooks typically focus on the initial development of new software products. However, at least for some such products, the initial development is but a progenitor of additional development to come. The Linux kernel is a large-scale and long-lived software system that is enjoying widespread use throughout the world. More than 800 releases have been made since version 1.0 was released in March 1994, and thousands of developers have contributed to the code base. The project is open source, and all the code of all the releases is freely available on the Internet (from www.kernel.org and numerous mirrors). This provides data for the study of software evolution of a nature and scale that is impossible with other, especially closed source, systems.

Operating systems such as Linux are examples of what Lehman has called E-type programs [28, 11]. They are part of the infrastructure of the world in which they are used. At the same time, they are modified in response to new requirements and change requests from users, evolving in ways that cannot be anticipated or specified in advance [15]. They thus undergo what we call *perpetual development*, with new releases of production versions occurring from time to time [45, 14]. While such development may of course come to an end when use of the system is discontinued, the mindset of the developers at any given time is that it will continue indefinitely.

Lehman's study of the evolution of E-type systems led to the formulation of a set of "laws" of software evolution. These include observations such as that software that is in constant use

inevitably continues to grow, to be adapted to its environment [28, 30]. Importantly, it was claimed that these laws are general and do not depend on the specific process being used to develop the studied software. The laws were validated by analyzing the evolution of size metrics of large-scale commercial systems such as the IBM OS/360. However, additional research has been hampered by the difficulty in obtaining high-quality data regarding large-scale commercial systems [23].

Today we can get data from open source products (e.g. [38, 7, 36]). While there is some debate on whether this also reflects the behavior of closed-source software, there are indications that it indeed does [44]. At the same time, the growth of the open source movement and the resulting importance of open source software to the industry as a whole mean that studying the evolution of open source is important in its own right, even if it does not reflect closed-source as well.

In this study we will use the Linux kernel. Our objective is to examine whether Lehman's laws are reflected in the development of the Linux kernel. A restricted version of this was done in the past by Godfrey and Tu [16], who examined the growth of the Linux kernel over its first six years (1994–2000), and found that at the system level its growth was superlinear. This result deviated from previous results obtained by Lehman using closed-source products [34, 32]. We want to verify this result using a much larger data set, and extend it to the other laws. Given the availability of the source code, we also use additional metrics that have not been used before in this context.

Linux is undoubtedly a successful open source system, which has been used for data in many previous studies (e.g. [3, 16, 51, 44, 64, 48, 20, 56, 58]). Specifically, we will analyze the progress made in hundreds of versions of Linux, released between March 1994 and August 2008, each containing thousands of source files and millions of lines of code. Such an analysis can serve as a basis for comparison with other systems. Linux is also interesting and important enough to be studied for its own sake.

The organization of this paper is as follows. Section 2 provides background regarding the Linux kernel and Lehman's laws of software evolution. In Section 3 we present our interpretation of these laws in the context of the Linux system, and derive quantitative data in their support. Section 4 concludes and includes a discussion of threats to validity and future work.

2. Background

2.1. Structure of the Linux Kernel

The Linux operating system kernel was originally developed by Linus Torvalds, who announced it on the Internet in August 1991. There followed $2\frac{1}{2}$ years of development by a growing community or developers, and the first production version was released in March 1994.

Initially a 3-digit system was used to identify releases. The first digit is the generation; this was 1 in 1994 and changed to 2 in 1996. The second digit is the major kernel version number. Importantly, a distinction was made between even and odd major kernel numbers: the even digits (1.0, 1.2, 2.0, 2.2, and 2.4) correspond to stable production versions, whereas versions with odd major numbers (1.1, 1.3, 2.1, 2.3, and 2.5) are development versions used to test new features and drivers leading up to the next stable version. The third digit is the minor kernel version number. New releases (with new minor numbers) of production versions supposedly included only bug fixes and security patches, whereas new releases of development versions included new (but not fully tested) functionality.

The problem with the above scheme was the long lag time until new functionality (and new drivers) found their way into production kernels, because this happened only on the next major version release. The scheme was therefore changed with the 2.6 kernel in December 2003. Initially, releases were managed at a relatively high rate. Then, with version 2.6.11, a fourth number was added. The third number now indicates new releases with added functionality, whereas the fourth number indicates bug fixes and security patches. Kernel 2.6 therefore acts as production and development rolled into one. However, it is actually more like a development version, because new functionality is released with relatively little testing. It has therefore been decided that version 2.6.16 will continue to be updated with bug fixes even beyond the release of subsequent versions, and act as a more stable production version[1].

The Linux kernel sources are arranged in several major subdirectories [49]. The main ones are

init contains the initialization code for the kernel.

kernel contains the architecture independent main kernel code.

mm contains the architecture independent memory management code.

fs contains the file system code.

net contains the networking code.

ipc contains the inter-process communication code.

arch contains all the kernel code which is specific for different architectures (for example Intel processors with 32 or 64 bits, PowerPC, etc.).

drivers contains all the system device drivers.

include contains most of the include (.h files) needed to build the kernel code.

An interesting note is that both the arch and drivers directories are practically external to the core of the kernel, and each specific kernel configuration uses only a small portion of these directories. However, these directories constitute a major part of the code, which grows not only due to improvements and addition of features, but also due to the need to support additional processors and devices.

2.2. Research Data

We examined all the Linux releases from March 1994 to August 2008. There are 810 releases in total (we only consider full releases, and ignore interim patches that were popular mainly in 1.x versions). This includes 144 production versions (1.0, 1.2, 2.0, 2.2, and 2.4), 429 development versions (1.1, 1.3, 2.1, 2.3, and 2.5), and 237 versions of 2.6 (up to release 2.6.25.11). This represents a significant extension of the work of Godfrey and Tu [16], who's cutoff date was January 2000 (their last versions were 2.2.14 and 2.3.39).

We aimed at examining the *entire* source code of the Linux Kernel. A typical C program consists of files whose names have two different extensions: .h and .c. The .c files consist of the

[1]This seems to have stopped with 2.6.16.62 released on 21 July 2008.

executable statements and the .h files of the declarations. Both should be looked at in order to get the full picture of the Linux code. In this we follow Godfrey and Tu [16] and Thomas [58], as opposed to Schach et al. [51] who only included the .c files. Given that we are interested in code evolution rather than in execution, we take the developer point of view and consider the full code base. Schach et al. [51] considered only the kernel subdirectory, that contains but a small fraction of the most basic code. Thomas [58] spent much effort on defining a consistent configuration that spans versions 2.0 to 2.4, thus excluding much of the code (mainly drivers), and limiting the study of innovations. In any case, such a consistent version cannot be found spanning the full range from 1.0 to 2.6. We ignore makefiles and configuration files, as was done by others.

We decided to examine both the development and the production versions as most of the evolution occurs in development versions. Thomas [58] in contradistinction examined only the second generation production series (2.0, 2.2, and 2.4). This precludes study of the development leading up to each new version, and of migration of code between development and production versions. Moreover, in retrospect it seems to be somewhat misguided as we believe that version 2.4 reflects significant development activity for at least a year (most of 2001). Godfrey and Tu [16] examined only a sample of the minor releases of each major version, and specifically decided to examine more production kernels, which were less frequently released. Such sampling precludes a detailed examination of the changes from one release to the next.

The code metrics that we measured and methodological issues are described in the appendix. To study the evolution of the system, we plot various metrics as a function of time, as suggested by Godfrey and Tu [16]. We then use a simple visual inspection to comment on observed patterns. The alternative is to use statistical tests, as was done for example by Lawrence [26]. However, the results of such statistical tests depend on the test used, and require a precise quantifiable definitions for Lehman's Laws — both of which are open to controversy. In addition we note that in many cases the overall observed behavior is erratic, sometimes including large discrete jumps. We therefore limit most of the conclusions to strong effects that are self-evident from the data.

2.3. Lehman's Laws of Software Evolution

Textbook lifecycle models divide a system's lifetime into two parts: development up to the initial release, and maintenance thereafter. But many real systems, and in particular E-type systems, are actually subject to perpetual development, and continue to evolve throughout their life. We are concerned with the study of how the system develops during its evolution.

Lehman studied several large scale systems and identified the following laws of software evolution, which describe behavioral patterns rather than technical issues [29, 30]. Empirical support for the laws was provided by several studies, typically using analysis of how software size changes with time [34, 31, 32]. We intend to follow a similar procedure, this time using Linux as the test case, and several different metrics. In particular, one of our contributions is the attempt to reduce the laws into a quantifiable form. In this we exploit the fact that we have access to all the source code, so various code metrics may be employed.

The laws and their implications are as follows, using the phrasing in [27]. The numbering reflects the order in which they were formulated.

1. *Continuing change (1974):* "An E-type system must be continually adapted, else it becomes progressively less satisfactory in use." Adaptation is to the circumstances of the system's

use; without it, the system will not keep up with the needs. For example, if business practices change, programs that support the business must change too. Specifically with E-type systems, using the program itself affects the environment, thereby driving the need for modifications. Thus a useful program is never finished, and resources must be allocated to its continued development.

2. *Increasing complexity(1974):* "As an E-type system is changed its complexity increases and becomes more difficult to evolve unless work is done to maintain and reduce the complexity." This complements the first law. When the program is changed, the first concern is the needed functionality. Thus the changes are typically done as a patch, disregarding the integrity of the original design. The implication is that in order to keep the program operational, it is not enough to invest in changing it — one also needs to invest in repeatedly reducing the complexity again to acceptable levels, in order to facilitate, or at least simplify, subsequent changes.

3. *Self regulation (1974):* "Global E-type system evolution if feedback regulated." This reflects a balance between forces that demand change, and constraints on what can actually be done. The rate of change is thus determined, among other factors, by the maintenance organization, and one must accept the limits this imposes. This law also has the important consequence that models of program evolution can be used as planning tools.

4. *Conservation of organizational stability (invariant work rate) (1980):* "The work rate of an organization evolving an E-type software system tends to be constant over the operational lifetime of that system or phases of that lifetime." This reflects the difficulty in moving staff, making budget changes, etc. Large organizations, that produce large software systems, have inertia, and tend to continue working in the same way. Trying to change this is hard and often proves futile.

5. *Conservation of familiarity (1980):* "In general, the incremental growth (growth rate trend) of E-type systems is constrained by the need to maintain familiarity." Maintaining familiarity is important for both developers and users. Thus large releases are typically followed by small ones that are required in order to restore stability. Trying to make too many changes at once is very likely to lead to serious problems.

6. *Continuing growth (1980):* "The functional capability of E-type systems must be continually enhanced to maintain user satisfaction over system lifetime." This is an extension of the first law: functionality is not only adjusted to changing conditions, but also augmented with totally new capabilities that reflect new demands from users, marketing people, and possibly others.

7. *Declining quality (1996):* "Unless rigorously adapted and evolved to take into account changes in the operational environment, the quality of an E-type system will appear to be declining." This is a corollary to the first law, and states the results of violating it. Specifically, if the software is not adapted, assumptions that were used in its construction will become progressively less valid, even invalid, and their latent effects unpredictable.

8. *Feedback system (1974/1996):* "E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems." Multiple participants are involved, including managers, develop-

ers, clients, and users. This is most explicitly covered by the third law, but feedback is also involved in the processes that determine other laws. This has to be recognized if one is to attempt to improve the process in order to achieve better efficiency.

These laws are not just a description of evolution, but rather an attempt to explain the processes driving the evolution. Thus they are not necessarily directly observable by examining the resulting software product. However, the software may reflect effects of the postulated laws. Lehman sought such effects in order to validate the laws, and we extend this quest to Linux.

3. Evolution of the Linux Kernel

In this section we analyze Lehman's laws of software evolution in light of data regarding the evolution of the Linux kernel. This is contrasted with Lehman's own work regarding empirical support for these laws [34, 31, 32]. Note that the laws are conventionally numbered by the date of their introduction. Here we use a different order, starting with those that are more directly related to the code, and grouping related laws together. Technical details regarding the measurements are given in the appendix.

3.1. Law 6: Continuing Growth

According to this law, functional content of a program must be continually increased to maintain user satisfaction over its lifetime. "Growth" can obviously also be interpreted as referring to the mere size of the software product. Moreover, the size can perhaps be used as a proxy for functional content, based on the assumption that additional code is written in order to support new features. Thus continuing growth may be validated by calculating software size metrics (such as number of modules) and tabulating their trends over time. This is the approach that was used by Lehman and others.

However, "Functional content" relates specifically to the software's features. In Linux and other operating systems the most direct metric for supported features is the number of system calls. This is augmented by the number of predefined parameter values that modulate the behavior of the system calls. For example, the well-know open system call has flags like O_CREAT to create the file if it does not exist, O_TRUNC to truncate the file if it does exist, or alternatively O_APPEND to add data to the end of the existing file, among others. Obviously adding such flags adds to the functionality of the system. We therefore also use such metrics in our study.

3.1.1. Functionality Metrics

Counting system calls in Linux is relatively easy, as they are listed in a table in file linux/arch/*/kernel/entry.S (somewhat surprisingly, there are system calls that are unique to different architectures, and we count all of them). We didn't also count flags, as manual pages have not been archived together with the code. Instead, we counted unique configuration options as they are listed in configuration files throughout the kernel. The results are shown in Fig. 1. In this and following figures, the X-axis is the release date, as suggested by Godfrey and Tu [16]. This is needed because there is significant variability in the development times represented by minor releases, so tabulating growth as a function of release number as suggested by Lehman might be misleading.

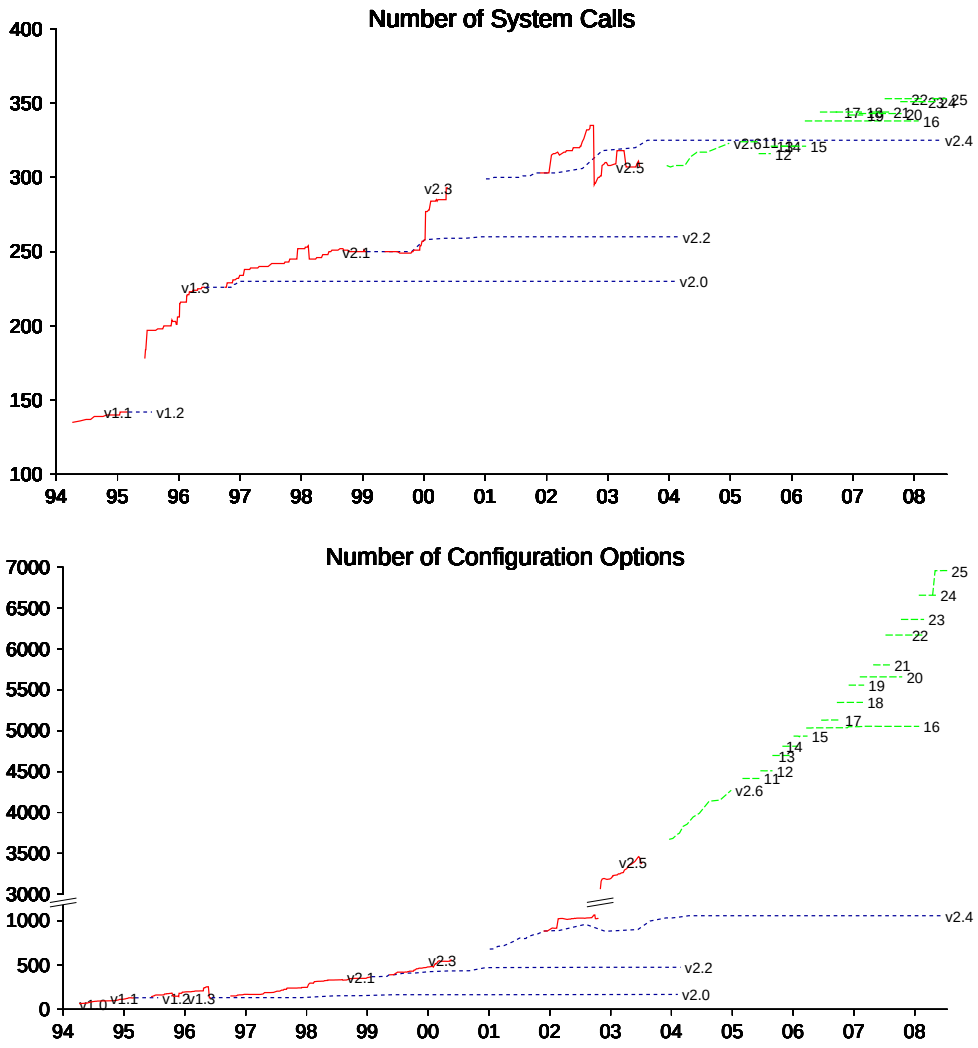

Figure 1: *The growing number of system calls and configuration options in Linux.*

The labels on the lines indicate the respective Linux kernel version; starting with 2.6.11 only the minor number is given.

According to our data, most system calls are added in development versions, with a 50% increase in 1.3 and another 20% rise in 2.3. Interestingly, in 2.5 the net effect was insignificant despite many additions, because a large group of OSF-related system calls were removed in version 2.5.42. The general rate of growth seems to be slowing down since 2003, probably indicating that the system is maturing. Configuration options, however, exhibit an opposite trend: they seem to be growing at an ever increasing rate. In fact, this is the only metric in which the growth in 2.6 seems to be superlinear. There is also a tripling of configuration options in version 2.5.45, which is probably technical in nature as it seems to be related to a change in their format and organization.

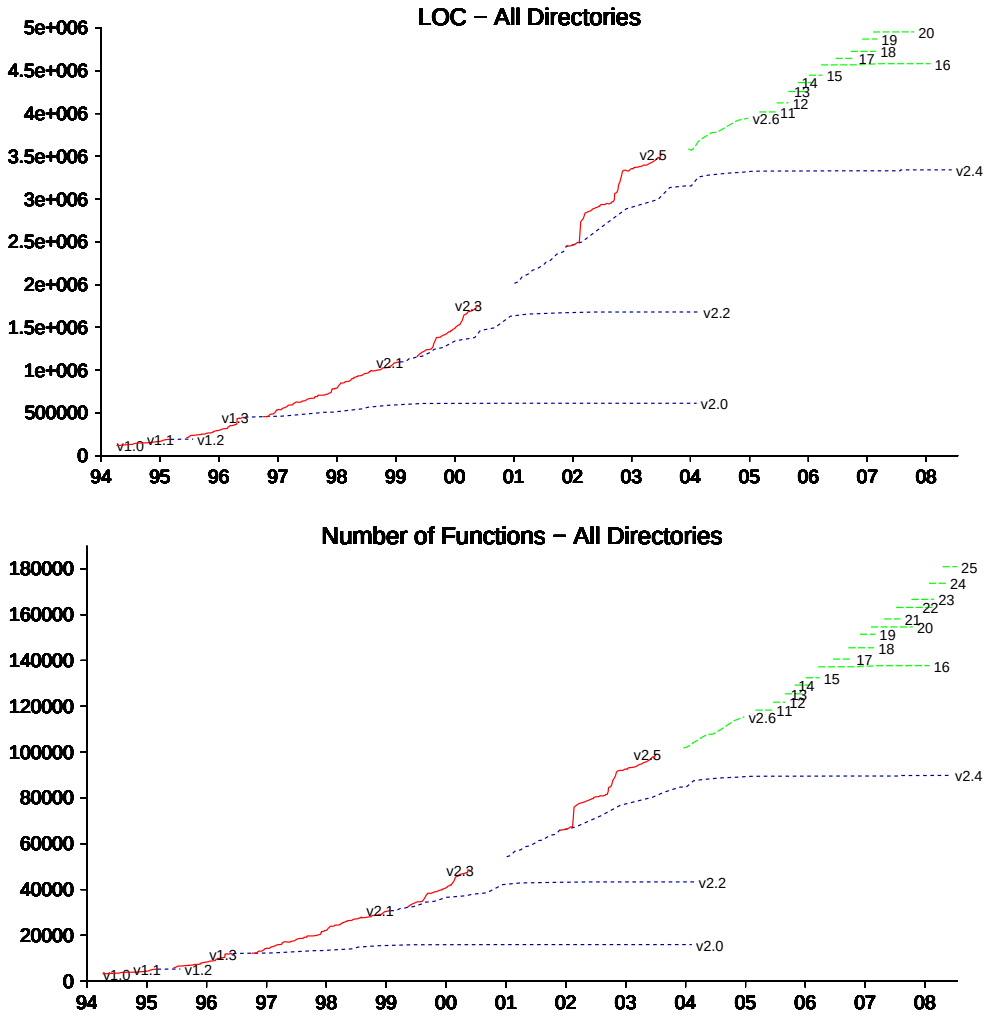

Figure 2: *The growth of Linux as measured by LOC and number of functions.*

3.1.2. Size Metrics

Various metrics may be used to describe the size of a software product. Among the most commonly used are lines of code (LOC) and number of modules. We will interpret lines of code as "non-comment non-blank" lines, i.e. we only count lines that actually contain some executable code. We interpret modules as C functions. Our results showing these size metrics for all Linux releases studied are shown in Fig. 2.

The two metrics, LOC and functions, lead to very similar results. As was expected, most of the growth occurs in development versions. However, there is significant growth also in the initial parts of the 2.4 and 2.2 production series. Interestingly, the growth curves sometimes exhibit discrete jumps, most notably in the 2.5 series. These can be traced to the inclusion of subsystems that were developed externally, such as the sound directory that was added in version 2.5.5, and the xfs file system that was added in 2.5.36. Such additions may be taken as examples of punctuated evolution

[12], as opposed to progressive evolution.

Comparing with previous work, Lehman, in his studies of closed-source systems, found that growth was either linear [31] or sub-linear [32], and in one case there seemed to be two distinct growth phases. His size metric was modules, and the independent variable was release number rather than time. This was analyzed by Turski who suggested an inverse-square law, where the increment in each release is equal to a constant effort divided by the square of the size of the previous release [60, 34]. The interpretation was that as size grows complexity increases, and it becomes harder to grow further: specifically, the inverse quadratic term reflects the need to consider all possible interactions. This model leads to the prediction that size would grow as the (sub-linear) cubic root of the release number.

Capiluppi analyzed several open-source projects, and found that growth is generally linear [6, 7]. In subsequent work using a larger dataset, Smith et al. found that some projects also exhibit periods of stability or even declining size, but still growth dominates in general [55]. Paulson et al. used a linear model for comparing the growth of open and closed software systems [44]. Godfrey and Tu [16] and Robles et al. [48], in contradistinction, found that Linux grows at a *super*linear rate, and suggested that a quadratic curve provides a good phenomenological model. Koch extended this to 8621 projects on SourceForge, and found that in general a quadratic model is better than a linear one, especially for large projects [24]. Note, however, that unlike Lehman's work this was based on counting LOC, not modules, and that growth was characterized as a function of time, not release number. Izurieta and Bieman compared Linux with FreeBSD, and concluded that both systems' growth rates are actually linear [20]. Scacchi reports on several studies that found a mix of sub-linear and super-linear growth in different open-source projects, including exponential growth [50]. Our results also support such a mix. When focusing on the most advanced codebase at any given time, as represented by the highest points in Fig. 2, the growth rates can be seen to be superlinear up to version 2.5, but in 2.6, which is beyond the data considered by most previous researchers, the growth seems to be closer to linear.

Interestingly, Torvalds predicted in 1999 that the growth of Linux would slow down as the system matures [59]. His reasoning was not related to complexity, but rather that the functionality required from the kernel will stabilize, and further development would take place in user space. The empirical data suggests that he was wrong.

3.2. Law 1: Continuing Change

According to this law, a program that is used must be continually adapted to changes in its usage environment else it becomes progressively less satisfactory. Usually, it is hard to distinguish between adaptation to the environment and general growth (as reflected in the continuing growth law) [32]. For example, when support for sound cards is added, is this a new feature or an adaptation to a changing environment? Most probably, it is both: a feature that was added in response to a change in the environment.

A special case that is relevant to Linux is adaptation to the changing hardware environment. Such changes are easily identifiable. All the code that pertains to processor architectures is contained in the arch subdirectory of the kernel. Likewise, all the code that pertains to peripherals is contained in the drivers subdirectory. Code that is added to these two subdirectories therefore reflects adaptation to the system's changing hardware environment. In fact, when looking at change

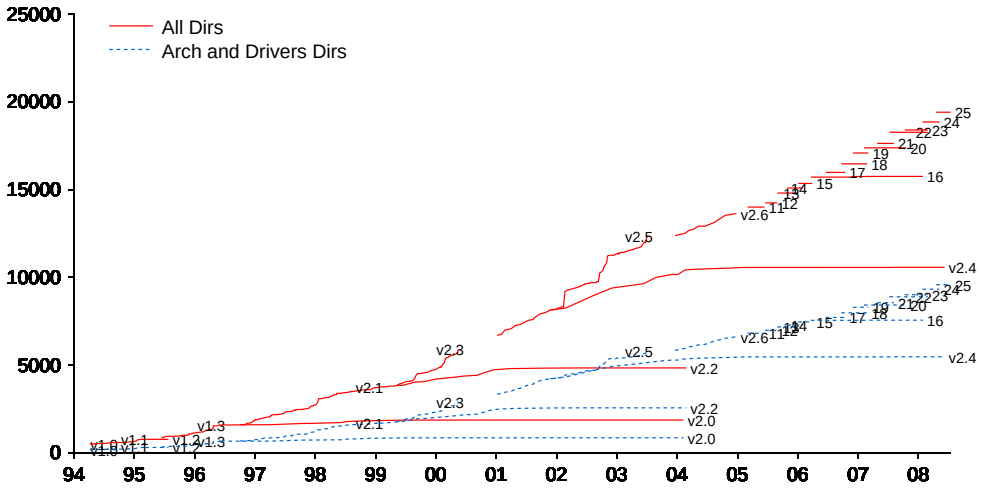

Figure 3: *The growth of source files in the* arch *and* drivers *directories as a fraction of the whole Linux system.*

logs, Linux forums, and explanations about the content of different versions, one of the things that returns in each version is new drivers and adaptation to new architectures.

A plot of how these two subdirectories grow with time is shown in Fig. 3. It is easily seen that they mirror the growth of the Linux kernel as a whole, and in fact account for about half of it in any given version. While the results shown are based on counting files, similar results are obtained when counting functions or LOC. In fact, the percentage of LOC contained in the arch and drivers directories is even a bit higher, and stands at about 60% on average. A possible explanation for this strong showing is that writing new drivers often involves cloning existing code [22].

We can thus assert the Linux exhibits continued change and adaptation to its environment, in accordance with this law, even though the original law is probably of wider scope than this specific example.

3.3. Law 2: Increasing Complexity

According to this law, as a program evolves its complexity increases unless work is done to maintain it by specifically making changes to reduce the complexity. This law is very hard to prove or disprove formally, as it allows both trends: if complexity increases it fits the initial premise of the law, but if it is reduced then maybe this is due to work that was done to reduce it, thus also satisfying the law. Moreover, work on extending the system typically includes work to keep it maintainable, and the two cannot realistically be separated [26].

Lehman supports this law by rationalization (adding features and devices necessarily increases complexity [29]) and by showing data that growth rates decline with time, as would be expected due to the constraints of increased complexity [34, 32]. However, our data regarding the Linux code growth rate, and the data of others as well [16, 48], does not display an inverse square pattern as claimed by Lehman. On the contrary, growth may actually be superlinear. Thus we do not see evidence that complexity is constraining the growth of Linux. And indeed, as we show below, there are indications that complexity is not increasing in Linux.

3.3.1. Direct Measurement of Code Complexity

An alternative approach is of course to measure code complexity directly, which is perfectly possible given that we have access to the full codebase of each version. In particular, we measured the McCabe cyclomatic complexity (MCC), which is equivalent to the number of conditional branches in the program plus 1 [35, 40]; for C, these include if-then-else, for, while, do, and case statements. We are aware of the fact that this metric has been challenged on both theoretical and experimental grounds, for example, by showing that it is strongly correlated with lines of code, or claiming that it only measures control flow complexity but not data flow complexity [53, 62, 54]. There is, however, no other complexity metric that enjoys wider acceptance and is free of such criticisms. Moreover, we do not attach much meaning to the absolute values cited, and are mainlyinterested in how the metric changes between successive versions of the same software product [17].

The results of calculating the MCC for the full codebase of all Linux versions are shown in Fig. 4. As may be expected, when the size of the code grows, so does the total MCC [53]. It is therefore more interesting to look at normalized values, such as the average MCC per function. The results in this case indicate a *declining* trend. Thus the total MCC is in general growing more slowly than the number of functions, and thus the average complexity is decreasing. A similar result is obtained if we normalize by LOC rather than by number of functions, except that MCC per LOC has been essentially stable since 2002, whereas MCC per function continues to decline. A decline is also observed in the median MCC across all the functions in the kernel. This was 4 in 1994, 3 from 1995 to the beginning of 2003, and 2 since then.

Focusing on the average MCC per function, one must remember that the number of functions also increases with time. Thus it might be that the reduced average value is just a result of having more functions with relatively lower complexity. Indeed, tabulating the average MCC in only new files each year leads to values that are typically lower than the average over all files [19]. We therefore looked at the distribution of the MCC per function over the different versions.

The distributions of major production versions are shown in Fig. 5. Each is represented by its first minor version; development versions are excluded because their initial releases tend to be very similar to the production version from which they branch. While the plots are qualitatively similar, one can observe that the order of the plots corresponds to that of the kernels: the innermost line is kernel 1.0, the next is 1.2, etc., and the topmost one is 2.6.25. Thus each line is more concave then the previous one. This means that over time we indeed have a larger fraction of functions with a lower MCC value.

Looking at the graphs more closely, one can observe that version 1.0 has significantly fewer functions with low MCC (in the range 1–6). This could imply that work was done specifically to reduce the complexity of the initial version code. Over the years, there was a significant improvement in general. For example, initially only about 38% of the functions had an MCC less than or equal to 2, but now it is about 52%. Using a threshold of 10, which was originally suggested by McCabe as indicating modules one should be worried about [35], and was used by the SEI to indicate “moderate risk” [57], we find that in 1.0 a full 15% of the functions had a higher MCC, but in recent versions this dropped to half — about 8%. This indicates that the vast majority of functions in Linux should be easy to maintain.

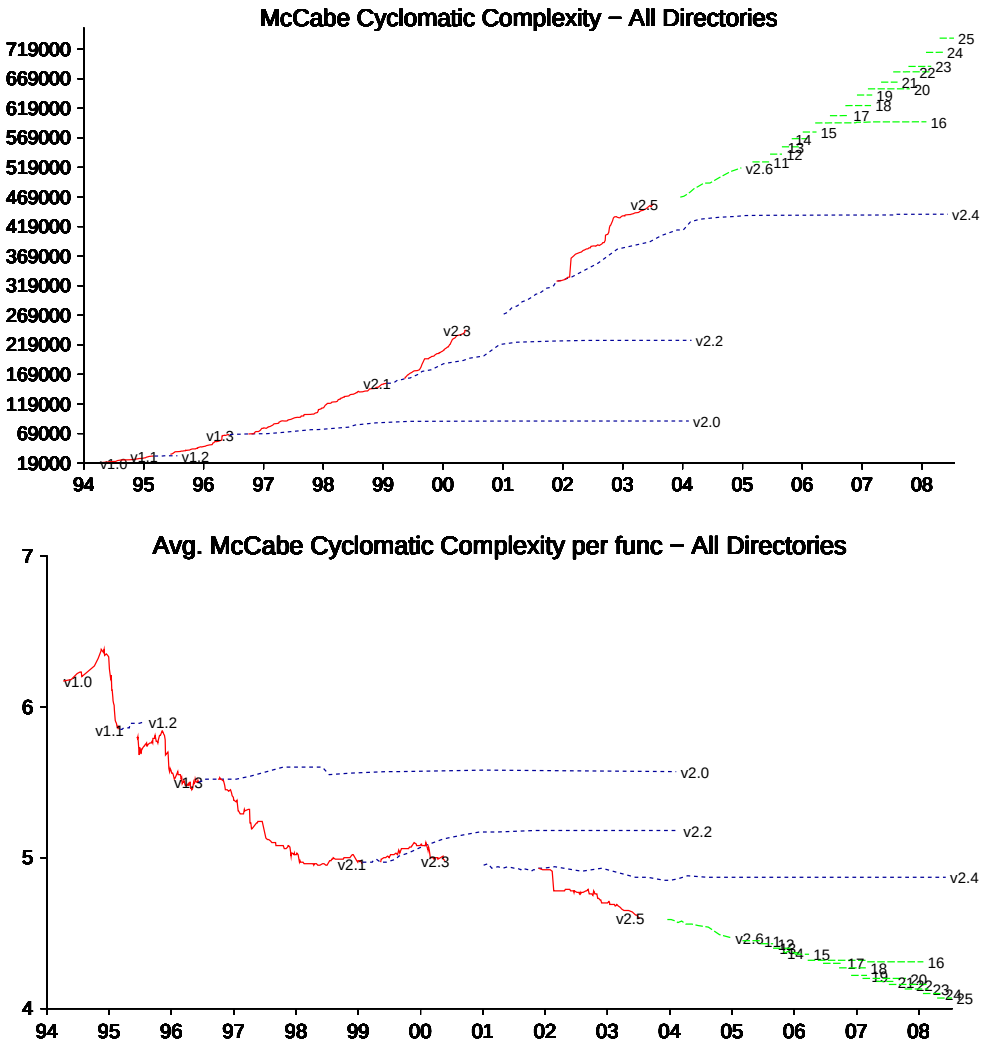

Figure 4: *Total McCabe's cyclomatic complexity and the average value per function.*

Another way to look at the distributions is to plot the evolution of their percentiles. This is shown in Fig. 6, this time using development versions (and 2.4 during its first year, when no contemporary development version existed). The results exhibit a dramatic decrease in the top percentiles, indicating that the fraction of functions with a high MCC is decreasing. This can mean that more low complexity functions are inserted, and/or that high complexity functions are being rewritten.

3.3.2. High-MCC Functions

While the results for the low-MCC functions are encouraging, one should also consider the high-MCC functions in the tail of the distribution. According to our results, 3–5% of the functions have an MCC above 20, which was classified by the SEI as "complex, high risk program" [57]; Microsoft's Visual Studio 2008 also reports a violation for values exceeding 25 [39]. The top

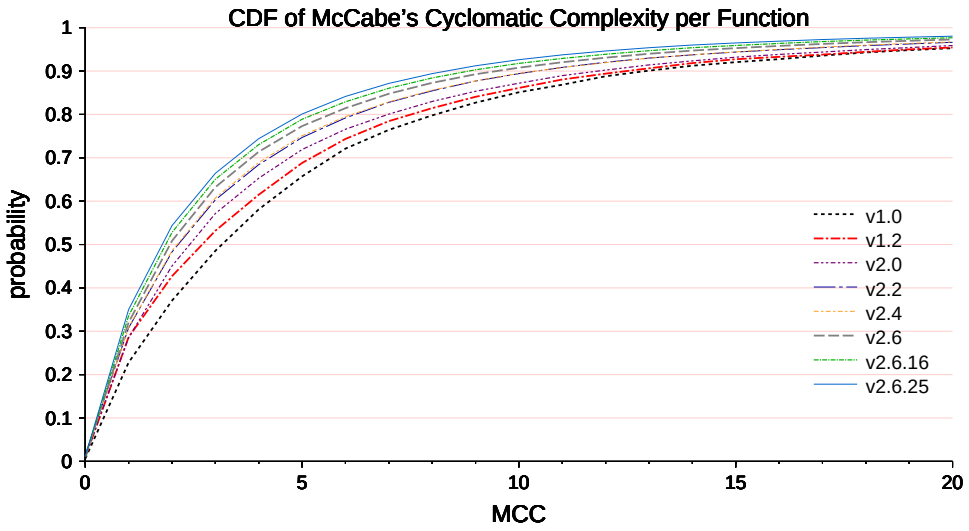

Figure 5: *Cumulative distribution function of McCabe's cyclomatic complexity per function for initial production versions.*

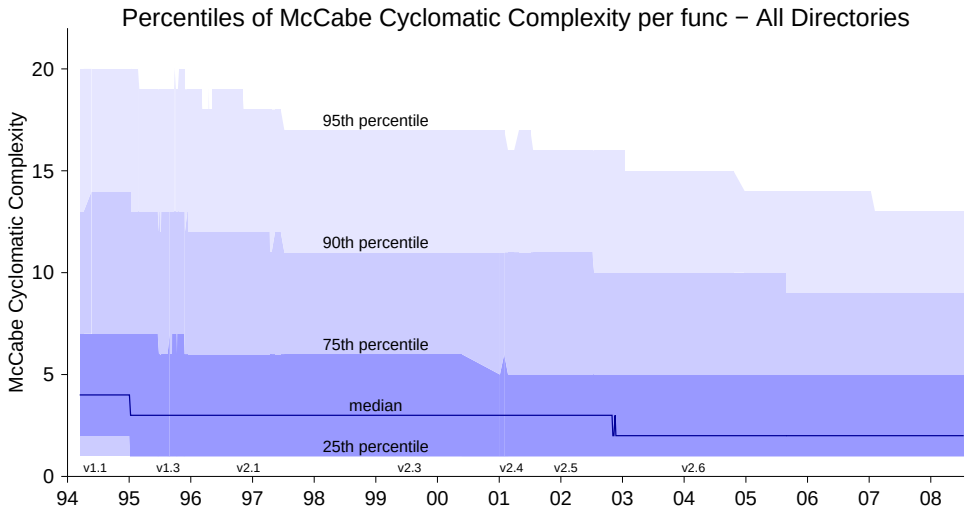

Figure 6: *Percentiles of McCabe's cyclomatic complexity per function for development versions.*

values observed are extremely high: for example, in version 2.6.16 we have a single function with MCC of 255; in version 2.2 there is a function with 470. Such values are totally out of the scale defined by the SEI, which classifies functions with an MCC above 50 as "untestable".

To study the evolution of the tail of the distribution, we plotted the survival function of the MCC values (i.e. for each MCC value, what is the number of functions that have higher value than that) in a log-log scale. Note that this is a variation on the conventional definition of the survival function, which uses the *fraction* of the functions that have higher values. Fig. 7 displays the results. The plots are close to being straight lines in the log-log axes, indicating a power-law

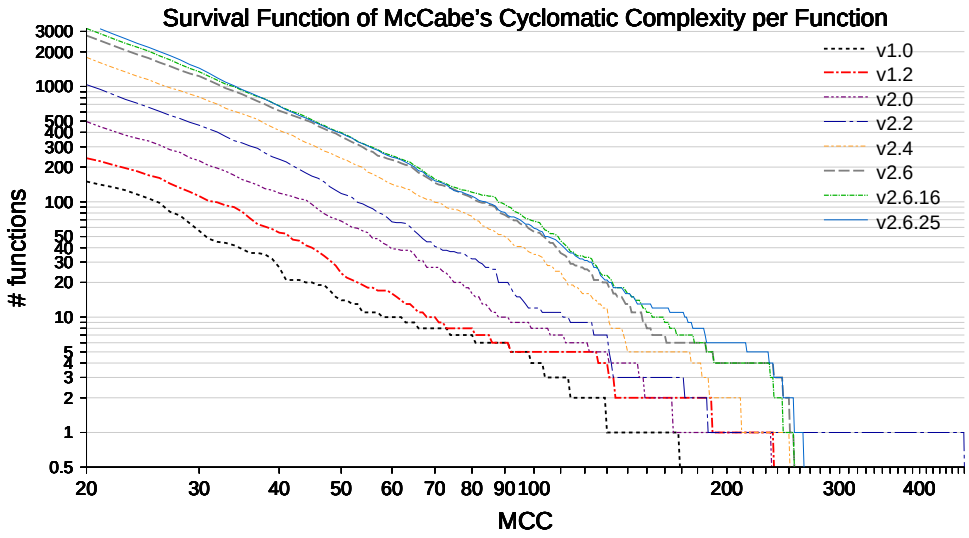

Figure 7: *Log-log survival functions of McCabe's cyclomatic complexity per function for selected kernels.*

relationship and that the distribution has a heavy tail. This could indicate that complexity is not added randomly, but rather that complex code tends to get more complex [10].

While the fraction of functions with high MCC is small and even diminishing (less than 5% have an MCC over 20, and less than 0.6% have an MCC over 50), the absolute number of such functions nevertheless grows considerably with time. For example, in 1.0 there were only 15 functions with an MCC of 50 or more. This grew to 25 functions in 1.2, 70 functions in 2.0, 100 functions in 2.2, 250 functions in 2.4, and 400 functions in 2.6/2.6.16.

The extremely high MCC values observed raises the question: what are these functions, and what happens to them over time? It turns out that the extreme MCC functions are usually from the arch and drivers directories. Typically, these functions are interrupt handlers or ioctl functions, which receive a request encoded as an integer, interpret it, and behave accordingly. The implementation is usually based on long switch statements with dozens of cases, or multiple if statements, leading to the extreme MCC values observed. We also found that many of the extreme MCC functions are the same for different kernels.

The highest MCC function in the whole dataset is sys32_ioctl from file arch/sparc64/kernel/ioctl32.c. As shown in Fig. 8, the file and function were introduced to the Linux kernel only in version 2.1.42 (in mid 1997). It grew both in LOC and MCC throughout versions 2.1 and 2.2, reaching an MCC value of almost 600 at the beginning of 2001. However, in version 2.3.47 (in the beginning of 2000) the LOC and MCC of this function dropped drastically to the MCC value of 6. It remained at that level throughout 2.4 and in most of 2.5, until it was completely removed (together with many other functions) in version 2.5.69 (mid 2003). The file itself was removed in 2.6.16.

The drop from an MCC value of around 600 to the value of 6 in version 2.3.47 was due to a major design change: instead of using the switch statement, a new compound data structure (a struct) with a handler was introduced. The request code was used as an index into a table of such structs, avoiding the need for multiple case statements. This is a real-life example of

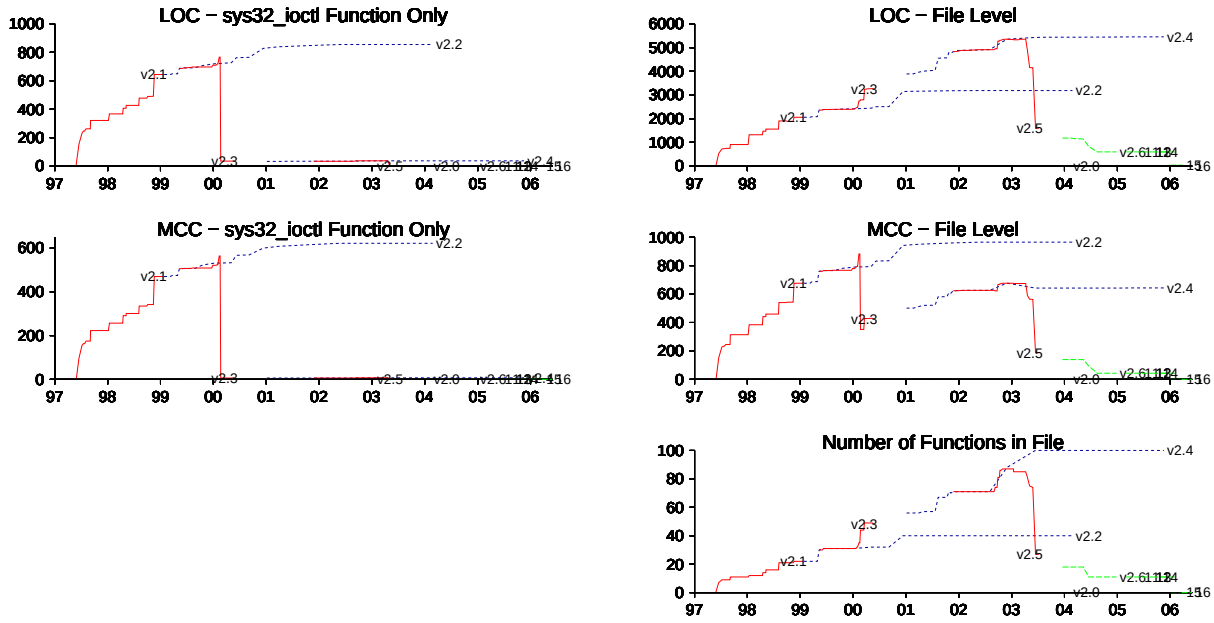

Figure 8: *Evolution of the* `sys32_ioctl` *high-MCC function.*

one of the critiques against the MCC metric: that the same functionality may be achieved either using `case` statements (which are counted) or using a table (which is not), both of which share similar complexity and testing difficulty [53]. However, it may be claimed that use of a table is indeed easier to comprehend and maintain, and therefore this is indeed an example of a significant reduction in code complexity.

In conclusion, we see evidence for an investment of work to reduce code complexity, both in version 1.1 and in specific high-MCC functions in later versions. Thus the general trend of reduced average MCC seems to result from a combination of code improvements and the addition of many low-MCC functions. Nevertheless, the number of high-MCC functions has also grown significantly.

3.4. Law 7: Declining Quality

According to this law, programs will be perceived as of declining quality unless rigorously maintained and adapted to a changing operational environment. Thus this law is somewhat of a corollary to Law 1, which demands continued change. It is also related to Law 2, which asserts increased complexity.

Another similarity to Law 2 is that this law is also actually impossible to prove or disprove, as it allows both trends (if quality declines, the law is supported, and if it is not, it might be due to the maintenance and adaptation efforts). Moreover, it is hard to measure “quality”. There are two main options: user perception and code metrics.

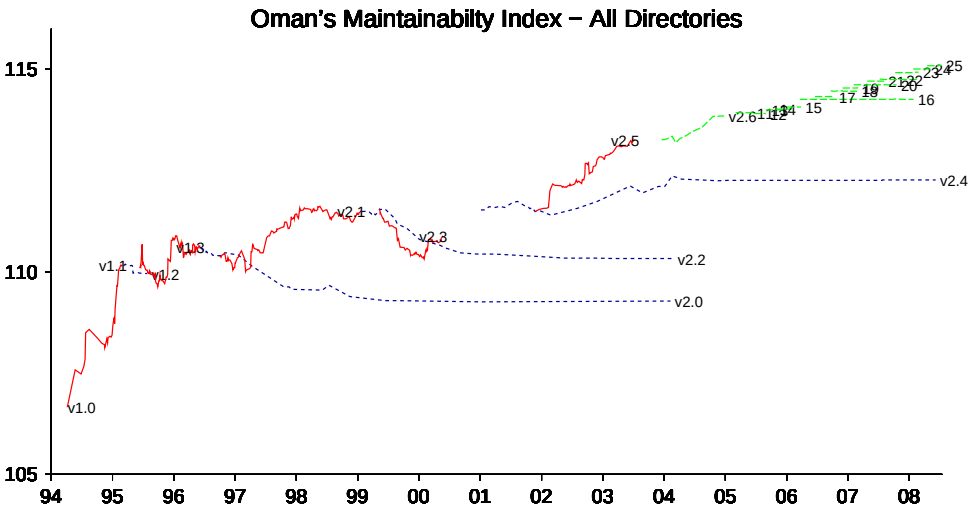

Figure 9: *Evolution of Oman's maintainability index.*

3.4.1. Perceived Quality

Lehman rationalizes this law by claiming that assumptions embedded in the system will tend to become inappropriate or even invalid with time [32], but does not offer measurable evidence. Similar rationalization is given by Parnas [43]. An expectable consequence of such a situation is that the system will fall out of use, because it will no longer fulfill its intended use or at least introduce inaccuracies or even errors.

Linux, on the other hand, has been in growing use for 14 years, and there are no indications that its adoption rate is abating. It is widely used on large-scale servers, clusters, and supercomputers. This may be taken as anecdotal indication that its quality is not declining, but rather that its usefulness is increasing. This increased usefulness may be partially explained by the effect of other pertinent laws, such as the continuing growth and the adaptation to the operational environment.

3.4.2. Measured Quality

As with software complexity, there is also no widely accepted metric for code quality. We decided to use the Oman maintainability index (MI) [42, 61] as a metric for quality, because it combines several other metrics that are generally agreed to be components of code quality: low complexity (as measured by MCC), small size (as measured by Halstead's volume (HV) and LOC), and good documentation (as measured by the fraction of LOC that contain comments). While the precise formula[2] used to calculate MI is debatable, being based on fitting data from a small number of not-too-large projects, this metric has nevertheless gained some following. Moreover, as with MCC, we are not interested in the actual values but only with how they change with time.

As MI is measured per module (or in our case, function), the data used is average LOC, MCC, and HV per function. In Linux, all these metrics decrease with time, thus contributing to a higher

[2] See the appendix for definition.

MI. The percentage of comment lines, on the other hand, has a slight downwards tendency. However the change is small, so we do not expect it to have a significant negative effect. As a result the general trend of MI is expected to be increasing with time, as is indeed seen in Fig. 9.

It is interesting to also dissect these results by directories (data not shown). It turns out that the sharp initial improvement in 1.1 is due to the core directories. The subsequent slower improvement has more to do with the arch and drivers directories. The lower values attributed to production versions are also due to these two directories. Another interesting point is that since the quality values for the core kernel directories are typically better than those of arch and drivers (i.e. less LOC, lower values for HV and MCC, and slightly more comments), we also see that the MI for these directories is somewhat higher — meaning that the core has slightly "better quality" than arch and drivers. This correlates with studies that point to drivers as a source of problems in operating systems [9, 1].

3.5. Law 4: Conservation of Organizational Stability (Invariant Work Rate)

According to this law, the average effective global rate of activity on an evolving system is invariant over the product life time. This measurement is technically problematic, since we are trying to look at "work" on the project. Reliable data about man hours or number of developers is hard to get in closed-source systems, and much harder (and maybe even ill-defined) in open-source projects. Moreover, man-hours are a notoriously inaccurate measure of work to begin with [5]. Lehman suggests using the number of elements handled (that is, added, deleted, or modified) as a proxy, but goes on to note that this too has methodological difficulties [32].

Taken at face value, this law is patently false for Linux. The number of people contributing to Linux's development has grown significantly over the years, and several companies now have employees assigned to working on it. This is reflected in the codebase growth rate noted above, which was superlinear at least through version 2.5. Thus it would seem that the rate of work on Linux has accelerated for at least the first 10 years. It is less clear whether the rate continues to grow now, when the growth rate seems to be linear rather than superlinear.

However, other interpretations also deserve to be considered. One is the number of elements handled, as suggested by Lehman. We will focus on development versions of Linux, where version releases are more frequent and reflect continuous activity by the developers (but in this case we also include the initial year of version 2.4, when there was no concurrent independent development activity). Fig. 10 shows the number of files that were added, deleted, or modified (divided into those that grew and those that shrunk) between successive releases. As may be expected, the absolute numbers tend to grow with time. But the *fraction* of files that are handled seems to be relatively stable, except perhaps for some decline in the first couple of years. Thus if we interpret "rate" to mean the fraction of the codebase that is modified in each release then the data supports the claim that the work rate is approximately constant. However, one should notice that the variance is high; such a high variance in a related metric — average handled modules per day — prompted Lawrence to claim that the rate is not constant at all [26].

Invariant work rate can also be interpreted with regard to the release rate itself, i.e. how often releases happen. In Fig. 11 we can see the number of releases per month, again using the development versions and for the initial period of 2.4. The results are that from mid 1996 to mid 2003 the rate seems stable at around 3–6 releases per month. We can see also that although version 2.4 has

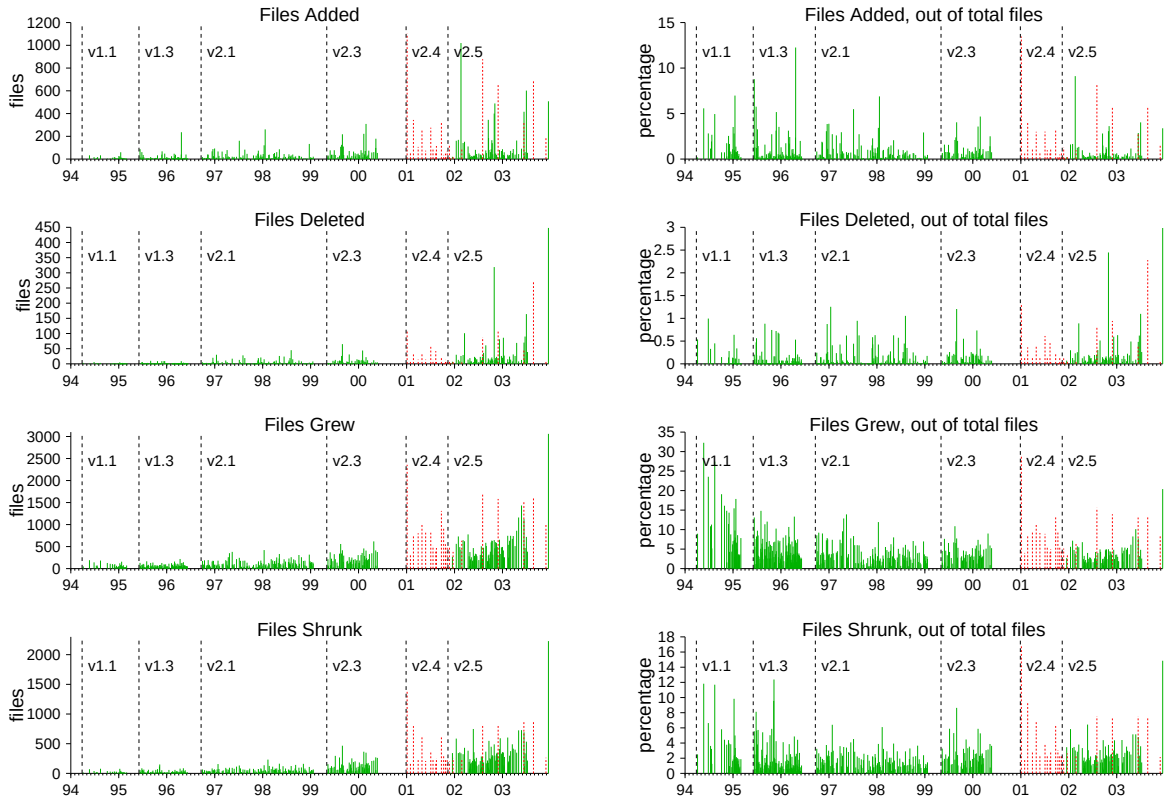

Figure 10: *Files added, deleted, grown, or shrunk among development versions.*

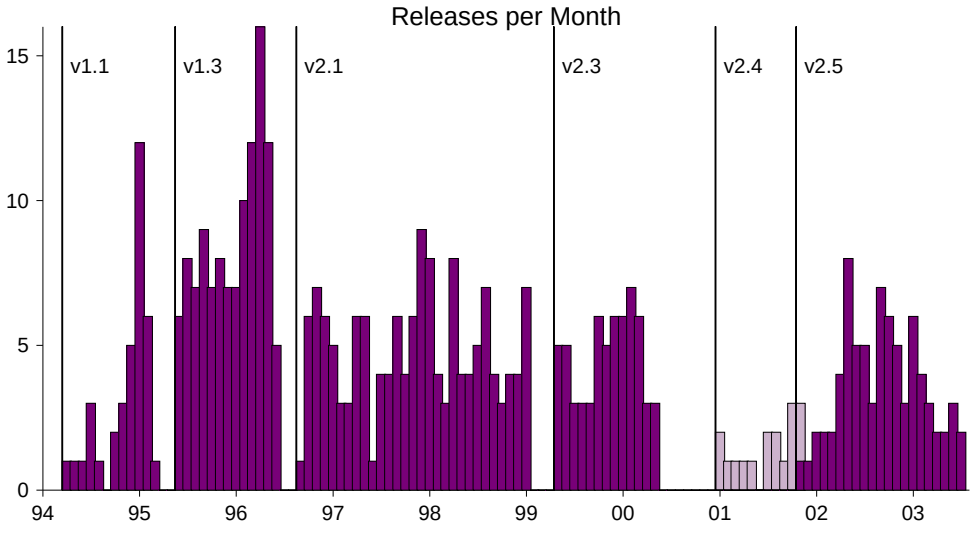

Figure 11: *Number of releases per month for development versions only.*

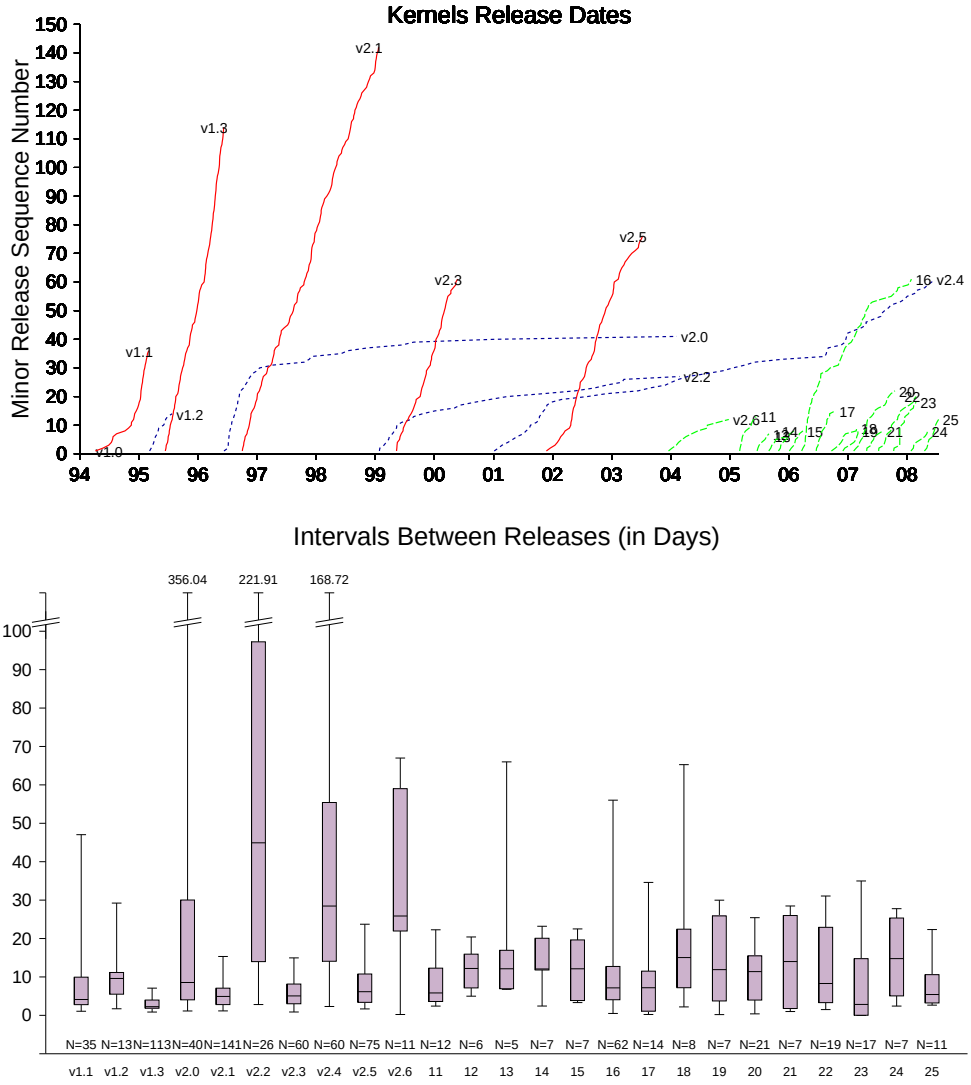

Figure 12: *Intervals between releases within major versions. Top: minor number as a function of time. Bottom: box plots show the* 5^{th}*,* 25^{th}*, median,* 75^{th}*, and* 95^{th} *percentiles, in days.* N *indicates the number of releases minus one, as the first release serves as "time 0".*

many features of development versions, it was released less frequently than the "real" development versions. Starting with version 2.6 in 2003, the 3^{rd} digit versions are timed to be released once every 2.5–3 months. This steady rate of releases also supports this law.

It is important to remember that Linux releases are organized into major and minor versions. Therefore one should consider the intervals between major releases separately from those leading to minor releases. The above data is shown again in Fig. 12, this time in the context of all other kernel versions. The top graph shows the times at which minor releases are made: the steeper the line, the higher the release rate [58]. Development versions exhibit a steady high rate, which reflects an invariant work rate. Production versions, however, tend to start with a somewhat high

rate and then taper off when the next development version is started. The box plots characterize the distribution of all the intervals for each major version. We see that development versions are consistently released at a high rate, as expected by the principle of "release early, release often" [47], and specifically all medians and 75th percentiles are lower than 10 days. Production version are released much less frequently, with median values that approach a month and 75th percentiles of two months and more.

All this changed with the new release scheme of 2.6. In the first 10 releases, the distribution was similar to that of previous production versions, but the tail of long release intervals was effectively eliminated. And with the 4^{th} digit releases of 2.6.11 and on, the whole distributions are generally much lower than in most production versions, albeit still higher than in the development versions. This should not be considered a problem, as 4^{th} digit releases are not development but rather bug fixes and security patches.

3.6. Law 5: Conservation of Familiarity

According to this law the change between successive releases is limited, both to allow developers to maintain their familiarity with the code, and to allow users to maintain their familiarity with using the system. Lehman et al. [34] suggest looking at the incremental growth — and if it is constant or declining on average, it indicates conservation of familiarity. Moreover, they suggest a threshold for which if two or more consecutive points exceeds it, the next points should be close to zero or negative. Lawrence claims that the series of incremental changes in the systems he checked was random, and interprets this as lack of conservation [26]. However, a better test may be the maximal change that occurred between successive versions.

In Linux conservation of familiarity with the code is reflected by the pattern of releasing development versions. As shown in Fig. 12, these releases come in rapid succession, typically only days apart. As shown in Figs. 2, 4, and 9, the development versions form a continuous line plotting the progress according to each metric, and production versions branch out directly from the last available development version. Taken together, these findings indicate that developers familiar with one version may expect little change in the subsequent ones. This stays the case even in cases where the system grew significantly due to the addition of some new module (e.g. the addition of the sound directory in version 2.5.5), because such additions are generally well encapsulated and do not have a strong effect on the rest of the system.

The effect of this law may also be apparent in the releases of new production versions. The most prominent example is the gap between the last 2.3 version and the initial 2.4 version. According to 2.4 logs and different Linux forums, this was a result of the version "not being ready" for release (according to Linus Torvalds's policy to release only when versions are stable), due to complications with development and testing many new features in that version. Thus it appears to be an instance of trying to put too many new features into a release, violating the conservation of familiarity law. Conversely, the new release scheme adopted for 2.6, where new production versions are released regularly every 2–3 months, may be viewed as an explicit attempt to limit the extent of new content in each release, in order to conserve familiarity.

Conservation of familiarity for users is specifically relevant when looking at successive stable production versions, which are the ones intended for end users. Our results indicate that in successive releases of the same major production version (or minor version in 2.6) the changes are very

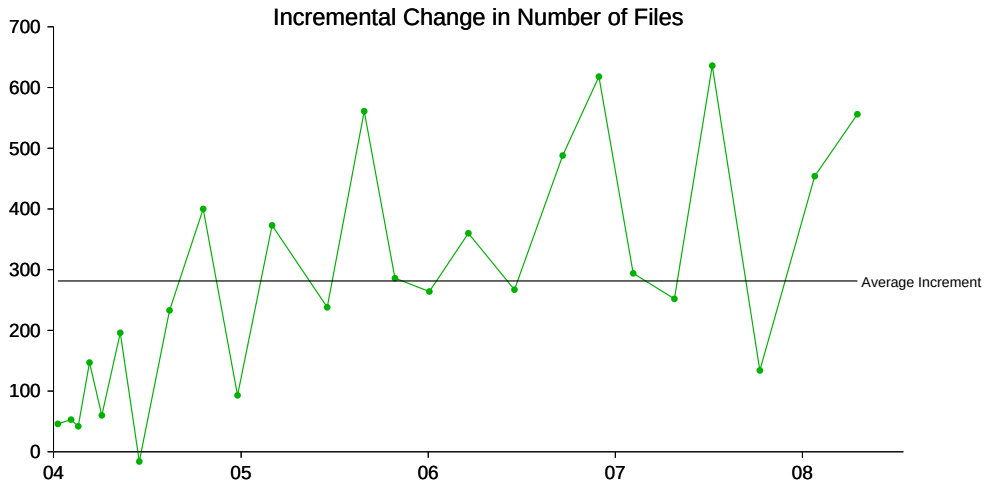

Figure 13: *Incremental growth of 2.6 versions.*

small (most of the time zero, but sometimes slightly increasing or decreasing). One can thus say that within production versions Linux indeed conserves user familiarity.

However, the changes between successive major versions are significant. In fact, they are so significant that users may opt to continue using an out-of-date previous release. This is witnessed by continued support for production versions long after the next one is released: in particular, note how 2.4 seems to track 2.6.16 in Fig. 12 (and of course users continue to use the system much after the last release). Thus we have both support for the law (as witnessed by the longevity of production versions) and contradiction of the law (because successive production versions with significant changes are nevertheless released).

3.7. Law 3: Self Regulation

According to this law, the program evolution process is self regulating, leading to a steady trend. Lehman finds evidence for this law in the fact that empirical growth curves show a ripple superimposed on a steady growth trend, and claims that the ripple demonstrates the interaction between the conflicting forces of desired growth and bounded resources [34, 32]. However, this interpretation may be challenged. An alternative one, which seems especially suitable for the Logica FW dataset, is that a larger growth occurs with the introduction of each new major release, followed by smaller growth during subsequent minor releases.

The existence of self regulation may be established by observing growth trends, where they imply that deviations from a consistently smooth growth will be corrected. The Linux dataset we use has many more releases than the datasets used by Lehman. Thus plotting growth leads to a continuous line, where individual releases are not seen (e.g. Fig. 2). The exhibited growth patterns are typically indeed quite steady, but do exhibit slight variations that may be considered as a ripple. But they also occasionally exhibit larger jumps as a result of integrating a new subsystem that was developed externally. The relatively smooth growth may be interpreted as resulting from self regulation, but it may also be the result of an invariant work rate.

We also performed an incremental growth analysis as was done by Lehman. We used data regarding 3rd digit release of version 2.6, where the release rate is relatively high and constant. Fig. 13 shows the change in the number of files (representing modules) per such release. The result is qualitatively similar to the observations of Lehman, namely that the growth rate seems to fluctuate around a mean, and that relatively large growth is nearly always followed by sub-average growth, indicating an alternation between growth and stabilization. Thus it is possible to interpret our data as providing indirect support for the existence of self regulation.

3.8. Law 8: Feedback System

The issue of feedback was mentioned already as an element of the self regulation law. A hypothetical example given by Lehman is pressure from the user community leading to more features and strong growth, followed by budgetary pressure limiting the testing and debugging capacity and thus reducing the growth rate again [34].

The claim that this self regulation stems from multi-level feedback is harder to establish. Lehman invested significant effort in supporting this law in the FEAST project, with the goal of improving software development processes [34, 33]. As we focus on characterizing the evolution of an existing system, this is beyond our scope.

Lehman supports this law by noting the stability of growth models, and in particular, that a handful of initial releases are enough to extract growth model parameters and predict subsequent sizes pretty accurately. However, he does note that this may also be largely due to organizational inertia [32].

In our specific case, Linux is the archetypal open-source system in which continued development is guided by feedback from the user community [47]. Examples range from defect reports, through bug fixes, and up to contribution of complete subsystems. Specific evidence for feedback affecting the software process itself is the switch to the 2.6 release scheme, in response to user discomfort with the long delays in releasing enhanced production versions. However, it is hard to bridge the gap from such observations to a quantitative law.

3.9. The Perpetual Development Lifecycle Model

Based on the above observations, it seems that E-type systems in general, and Linux in particular, conform to a lifecycle model that may be called “perpetual development”. This lifecycle model comprises the following elements.

- Continuous and steady development of the system, adding new features all the time [14]. Linux and other open source projects make this activity public, whereas in proprietary closed projects it is done behind the scenes. The development is done based on anticipated user needs and explicit user feedback, rather than preconcieved specifications of how the system should be used.
- When significant new functionality accumulates, the continuous development is interrupted to prepare a major release of a new production version. In Linux, the interval between such releases used to be more than a year, but was then reduced to 2–3 months in the 2.6 series.
- More common minor releases of an existing production version, reflecting bug fixes and security patches. Several production versions may be thus supported in parallel.

This is quite different from textbook lifecycle models, be they "one-shot" models like the waterfall model or iterative models like the spiral model.

As we have seen, articulating this model enables us better fidelity regarding some of Lehman's laws. For example, conservation of familiarity within major production versions is replaced by discontinuities between such major versions. It also has implications for software development in general, and in fact lies at the basis of many agile methods.

4. Conclusions

The Linux kernel is one of the most successful open-source software projects in the world. Over the last 14 years it has continued to evolve in order to satisfy the needs of its users. We have presented a detailed characterization of this process, including over 800 versions which represent new developments, major production releases, and minor updates. Many interesting phenomena are only seen at this fine resolution, and would be lost if using the traditional approach of studying only major production releases.

The study presented here was based on Lehman's Laws of software evolution. We found obvious support for continuing growth and change, and probable support for invariant work rate. Conservation of familiarity seems to be combined with large changes when new production versions are released. The practice of preventative maintenance seems to support the increasing complexity and declining quality laws, which note the possibility of work being done to prevent them. The hardest laws to justify are the self regulation and feedback system laws, for which we find only some anecdotal evidence (but there is a good case and justification for further investigation). The laws and our results are summarized in Table 1.

Taking a more global view of Linux's evolution, we find it to be a prime example of *perpetual development* — a system that is developed continuously in collaboration with its users, without elaborate specifications and planning. This is seen in the continuous trends observed along the backbone of development versions, and in the relative stability of production versions that branch out from this backbone.

The above observations are based on our interpretations of the results of an automatic analysis of the code. In particular, in many cases we suggest novel quantifications of Lehman's laws. Other quantifications are possible, representing a potential internal threat to the validity of our results. In some cases it is hard to arrive at a conclusion regarding what were the forces motivating the observed behavior. A further investigation of supplementary data, such as change logs and developer forums, is required in order to resolve those issues and give sufficient explanations to the phenomena. Also, the study of the code itself can be improved with the help of tools to comprehend and analyze the code structure [2, 46].

Our results are of course specific to Linux, thus representing an external threat to validity. But some observations may generalize to other software systems as well. The applicability and generality of our results can be assessed by replicating the study for other operating system kernels and for other large software (as in [56]), and by comparing the trends and the qualitative results of each. It might also be relevant to perform this comparison for open and closed source software in order to understand the differences in the development (as in [44]). The problem, of course, is to obtain suitable data for such a study.

No.	Lehman's Law	Manifestation in Linux
1	Continuing change	The arch and drivers directories, which account for 50–60% of the codebase, grow with the rest of the kernel, reflecting continued adaptation to the hardware environment
2	Increasing complexity (unless prevented)	While overall complexity grows with the code, the average per function is declining; in specific instances work to reduce complexity is evident
3	Self regulation	Possibly supported by steady overall growth rates and fluctuation of incremental growth, but there is no direct support for a regulation mechanism
4	Conservation of organizational stability (invariant work rate)	Rate of releases has been relatively stable from 1997 to 2003. The 2.6 method of timed releases also creates an invariant amount of releases per time unit
5	Conservation of familiarity	Long-lived production versions reflect this law — successive minor releases have little functionality changes. But there are big changes between successive production versions
6	Continuing growth	Growth in functionality is obvious. Growth of LOC and functions occurred at a super-linear rate up to version 2.5, and then closer to linear
7	Declining quality (unless prevented)	Declining quality is contradicted by increasing usefulness to users, and by consistent improvement in a composite maintainability metric. Thus continued work on Linux has prevented decline
8	Feedback system	Anecdotal support based on the structure of open source development

Table 1: *Support for Lehman's Laws in Linux.*

This study can also be extended by following additional metrics, such as the various types of common coupling [41, 64], the indirect metrics developed by Yu [63], the shape of the code tree [7], and more process-related metrics such the time spent, the number of people involved, and how many developers participate in each type of activity (some preliminary data is available in [25]). As the Linux kernel is very big, it would also be beneficial to perform a more detailed study of specific subsystems independently. This might allow us to better characterize and quantify the different laws of evolution, and arrive at more precise formulations. For example, the laws regarding increasing complexity and reduced quality beg for deeper study, and in particular the widespread identification and characterization of maintenance activity intended to reduce complexity and improve quality.

On a different trajectory, it would be interesting to complement the empirical characterization with more theoretical reasoning, in an attempt to uncover the forces at work. This is in fact the basis for some of Lehman's laws, e.g. self regulation, feedback system, and conservation of familiarity.

An example requiring additional study is the idea that invariant work rate implies that at each step in the evolution, the system will either grow or undergo some reorganization, but not both at once [12].

Acknowledgments

This work was supported by the Dr. Edgar Levin Endowment Fund. Many thanks to prof. Manny Lehman for commenting on a draft of this paper.

A. Methodology

Our analysis is based on measuring the different properties and metrics of the Linux kernel as described in [19]. For completeness, we repeat this here.

A.1. Software Metrics

Many different quantitative software metrics have been proposed over the years [21, 37]. These can be classified as product metrics which measure the software product itself (such as size as reflected by LOC and percentage of documented lines) and process metrics which measure the development process (such as development time and experience of the programming staff). We focus mainly on product metrics as these can be extracted reliably directly from the code.

There are two reasons why it is important to perform code-based measurements. The first is accuracy, as the alternatives of using surveys and logs can be highly inaccurate. For example, a survey of maintenance managers yielded the result that 17.4 percent of maintenance is corrective in nature, while a separate study based on analyzing changes to source code led to a result three times larger [52]. Similarly, a comparison between change logs for three software products and the corresponding changed source code itself showed that up to 80% of changes made to the source code were omitted from change logs [8].

The second reason why code-based metrics are important is that certain phenomena can be measured only by examining the code itself. For example, common coupling has been validated as a measure of maintainability [4], and the only way to measure the common coupling within a software product is to examine the code itself.

The metrics we measure are the following:

1. Number of modules, as expressed by the number of directories, files, and functions.
2. Lines of code (LOC), including its related variants: comment lines and total lines.
3. McCabe's cyclomatic complexity (MCC), which is equivalent to the number of conditional branches in the program plus 1 [35, 40]. These include if-then-else, for, while, do, and case statements. We also measure the extended version (EMCC) where one counts the actual conditions and not only the conditional statements, based on the conception that Boolean connectives add complexity to the code.
4. Metrics defined as part of Halstead's software science [18]. The building blocks of these metrics are the total number of operators N_1 and the number of unique operators n_1, as well as the total number of operands N_2 and the number of unique operands n_2. Using them, Halstead defined the following:

The volume $HV = (N_1 + N_2) \lg_2 (n_1 + n_2)$. This actually measures the total number of bits needed to write the program.

The difficulty $HD = \frac{n_1}{2} \cdot \frac{N_2}{n_2}$. This is proportional to the available tools (operators) and the average usage of operands.

The effort $HE = V \cdot D$. This is simply the product of how much code there is and the difficulty of producing it.

In cases when the metrics are undefined (e.g. for an empty function $n_1 = n_2 = 0$) they were taken as 0. This happened in around 1% of the functions.

5. Oman's maintainability index (MI) [42, 61], which is a composite metric that attempts to fit data from several software projects. Its definition is

$$MI = 171 - 5.2 \ln(\overline{HV}) - 0.23\overline{MCC} - 16.2 \ln(\overline{LOC}) + 50 \sin(\sqrt{2.46\overline{pCM}})$$

where $\overline{X}$ denotes the average of X over all modules, and pCM denotes the percentage of lines that are comments. However, following Thomas [58], we will interpret pCM as a fraction (between 0 and 1) rather than as a percentage (between 0 and 100) because with this interpretation $\sqrt{2.46\overline{pCM}}$ has the range of 0 to approximately $\frac{\pi}{2}$.

6. Files and directories handled (added, deleted, or modified).
7. The rate of releasing new versions.

A notable omission from the above list is common coupling, which has been used to assess the Linux code in several previous studies [51, 64, 58]. However, all those studies neglected to fully follow inter-procedural pointer dereferences, and thus potentially miss many instances of coupling. As this is an extremely difficult issue, we leave its resolution to future work.

A.2. Analysis Tool

In order to analyze our full dataset, a static analysis tool was required. We initially considered using a commercial CASE tool, but in the end developed a tool of our own that was simpler and did precisely what we wanted.

Linux is written in C, which includes pre-processor directives (#define, #ifdef, etc.) that are processed before the compilation proper. One of the big challenges we faced was to handle such directives. Generally there are two approaches to perform static code analysis: either to pre-process the code or not to. Applying pre-processing is useful when the goal is to understand how the code actually runs. However, our objective is to study software engineering, and in particular the evolution and maintenance of the code. Thus we believe that the correct way to go is to analyze the code as the *developer* views it, i.e. before applying the pre-processor.

For example, macros may be used as an aid to abstraction that has lower cost than subroutines because the code is inlined by the pre-processor. Due to the inlining, if a developer uses the macro #define MAX(X,Y) (X>Y)?X:Y, the "real" complexity of the code increases because a branch is added in each use. But from the developer's point of view this added complexity is hidden, and therefore should not be counted. Thus macros should be calculated with all their properties upon their definition, but as a function call in their uses. The same applies for #include, which can induce considerable bloat.

A major problem comes from use of conditional compilation directives such as #ifdef. Such code sections are used mainly to handle different configurations with essentially the same code, by singling out only the differences and selectively compiling the correct version for each configuration. This implies that if we use pre-processing we will actually only analyze a certain configuration and not the whole code base.

Assuming the developer's perspective again, we want to analyze *all* the code — all the versions of the #if directives, and their #else parts too — because a developer maintaining such a file must be aware of all different possibilities of the flow of the code. This again implies that pre-processing is inappropriate for our needs. In this we differ from Thomas [58], who used a CASE tool which requires pre-processing, and thus only files and code sections in the pre-processed configuration were examined.

The major problem with not performing any pre-processing is that the resulting code is not always syntactically legal. For example, a function may have slightly different signatures in different configuration, and this can be expressed using #ifdef and #else. With pre-processing, the compiler will only see the correct definition each time. But if we just delete the pre-processing directives, we will get two contradicting definitions of the same function one after the other, and moreover, sharing the same function body.

As it turns out, in most cases we were able to analyze files which had #ifdef sections. In other cases, we saw that when analyzing both paths of the #if a malformed code is created and thus we were not able to analyze it with our automated tool. Trying to do this manually is also a challenge — how does one decide which path to analyze? Therefore, when we encountered such files we removed them from the analysis. Other malformed files (very few) were removed as well and are not included in our calculations.

Overall, less than 1.5% of the source files were not analyzed at all. Among the arch and drivers subdirectories between 0.3%–3% of the files were not analyzed, whereas in the other parts of the kernel the worst case of un-analyzed files was less than 0.7%. Thus the vast majority of the files were analyzed and their data is aggregated in the different metrics.

Our analysis tool, while not free of problems and limitations, is tailored to perform the analysis based on the above considerations. We coded a perl script that, given a C file, parses it into its different tokens, and generates an output file with the metrics. Pre-processor directives were stripped out, and all the remaining code was analyzed. In those cases where this practice led to inconsistent code the file was removed from consideration as explained above. Empty functions and files were not considered problematic and are included in the metrics, as they are part of the design.

We ran this program on all the .c and .h files of all the versions, creating an output file with the calculated metrics for each one. Then, for each version we aggregated all the data from these output files. This was done for three groupings: the whole kernel, only the arch and drivers directories, and only the other (core) directories. This allows us to study whether these subsystems behave differently from each other and from the whole system [13, 16].

In order to aggregate the metrics at the kernel level, we used the same approach used in other studies (such as Thomas [58]) and as explained in the original metrics definitions. For example, LOC, MCC, and EMCC are simply summed across all files (note that files with no functions, such

as some header files, will have an MCC of 0, because MCC is a function level metric). In order to compare the different versions, despite the addition of new files or functions, we sometimes look at the average metric values of the files and functions of the kernel rather than at the aggregate values. The same is true about the function-level Halstead metrics. Oman's MI is defined at the file level. Since it has a 100-point scale we cannot aggregate the values. Instead, we will use only the average the MI values of all files in the kernel.

References

[1] A. Albinet, J. Arlat, and J.-C. Fabre, "*Characterization of the impact of faulty drivers on the robustness of the Linux kernel*". In *Intl. Conf. Dependable Syst. & Networks*, pp. 867–876, Jun 2004.

[2] T. Ball and S. G. Eick, "*Software visualization in the large*". *Computer* **29(4)**, pp. 33–43, Apr 1996.

[3] I. T. Bowman, R. C. Holt, and N. V. Brewster, "*Linux as a case study: Its extracted software architecture*". In 21st *Intl. Conf. Softw. Eng.*, pp. 555–563, May 1999.

[4] L. C. Briand, J. Wust, and H. Lounis, "*Using coupling measurement for impact analysis in object-oriented systems*". In *Intl. Conf. Softw. Maintenance*, pp. 475–482, Aug 1999.

[5] F. P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1975.

[6] A. Capiluppi, "*Models for the evolution of OS projects*". In *Intl. Conf. Softw. Maintenance*, pp. 65–74, Sep 2003.

[7] A. Capiluppi, M. Morisio, and J. F. Ramil, "*Structural evolution of an open source system: A case study*". In 12th *IEEE Intl. Workshop Program Comprehension*, pp. 172–182, Jun 2004.

[8] K. Chen, S. R. Schach, L. Yu, J. Offutt, and G. Z. Heller, "*Open-source change logs*". *Empirical Softw. Eng.* **9**, pp. 197–210, 2004.

[9] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "*An empirical study of operating system errors*". In 18th *Symp. Operating Systems Principles*, pp. 73–88, Oct 2001.

[10] G. Concas, M. Marchesi, S. Pinna, and N. Serra, "*Power-laws in a large object-oriented software system*". *IEEE Trans. Softw. Eng.* **33(10)**, pp. 687–708, Oct 2007.

[11] S. Cook, R. Harrison, M. M. Lehman, and P. Wernick, "*Evolution in software systems: Foundations of the SPE classification scheme*". *J. Softw. Maintenance & Evolution: Res. & Pract.* **18(1)**, pp. 1–35, Jan-Feb 2006.

[12] S. Cook, R. Harrison, and P. Wernick, "*A simulation model of self-organising evolvability in software systems*". In *IEEE Intl. Workshop Software Evolvability*, pp. 17–22, Sep 2005.

[13] H. Gall, M. Jazayeri, R. R. Klösch, and G. Trausmuth, "*Software evolution observations based on product release history*". In *Intl. Conf. Softw. Maintenance*, pp. 160–166, Oct 1997.

[14] L. Gasser, W. Scacchi, G. Ripoche, and B. Penne, "*Understanding continuous design in F/OSS projects*". In 16th *Intl. Conf. Softw. & Syst. Eng. & Apps.*, Dec 2003.

[15] M. W. Godfrey and D. M. German, "*The past, present, and future of software evolution*". In 24th *Intl. Conf. Softw. Maintenance*, Sep 2008. (Special track on Frontiers of Software Maintenance).

[16] M. W. Godfrey and Q. Tu, "*Evolution in open source software: A case study*". In 16th *Intl. Conf. Softw. Maintenance*, pp. 131–142, Oct 2000.

[17] G. A. Hall and J. C. Munson, "*Software evolution: Code delta and code churn*". *J. Syst. & Softw.* **54(2)**, pp. 111–118, Oct 2000.

[18] M. Halstead, *Elements of Software Science*. Elsevier Science Inc., 1977.

[19] A. Israeli and D. G. Feitelson, "*Characterizing software maintenance categories using the Linux kernel*", Feb 2009. Submitted for publication.

[20] C. Izurieta and J. Bieman, "*The evolution of FreeBSD and Linux*". In 5th *Intl. Symp. Empirical Softw. Eng.*, pp. 204–211, Sep 2006.

[21] S. H. Kan, *Metrics and Models in Software Quality Engineering*. Addison Wesley, 2nd ed., 2004.

[22] C. J. Kasper and M. W. Godfrey, "*"Cloning considered harmful" considered harmful: Patterns of cloning in software*". *Empirical Softw. Eng.* **13(6)**, pp. 645–692, Dec 2008.

[23] C. F. Kemerer and S. Slaughter, "*An empirical approach to studying software evolution*". *IEEE Trans. Softw. Eng.* **25(4)**, pp. 493–509, Jul/Aug 1999.

[24] S. Koch, "*Evolution of open source software systems – a large-scale investigation*". In 1st *Intl. Conf. Open Source Systems*, pp. 148–153, Jul 2005.

[25] G. Kroah-Hartman, J. Corbet, and A. McPherson, *Linux Kernel Development — How Fast is it Going, Who is Doing it, What are they Doing, and Who is Sponsoring it*. Tech. rep., the Linux Foundation, Apr 2004.

[26] M. J. Lawrence, "*An examination of evolution dynamics*". In 6th *Intl. Conf. Softw. Eng.*, pp. 188–196, Sep 1982.

[27] M. Lehman and J. C. Fernández-Ramil, "*Software evolution*". In *Software Evolution and Feedback: Theory and Practice*, N. H. Madhavji, J. Fernández-Ramil, and D. E. Perry (eds.), chap. 1, pp. 7–40, Wiley, 2006.

[28] M. M. Lehman, "*Programs, life cycles, and laws of software evolution*". *Proc. IEEE* **68(9)**, pp. 1060–1076, Sep 1980.

[29] M. M. Lehman, "*On understanding laws, evolution, and conservation in the large-program life cycle*". *J. Syst. & Softw.* **1**, pp. 213–221, 1980.

[30] M. M. Lehman, "*Laws of software evolution revisited*". In 5th *European Workshop on Software Process Technology*, pp. 108–124, Springer Verlag, Oct 1996. Lect. Notes Comput. Sci. vol. 1149.

[31] M. M. Lehman, D. E. Perry, and J. F. Ramil, "*Implications of evolution metrics on software maintenance*". In 14th *Intl. Conf. Softw. Maintenance*, pp. 208–217, Nov 1998.

[32] M. M. Lehman, D. E. Perry, and J. F. Ramil, "*On evidence supporting the FEAST hypothesis and the laws of software evolution*". In *Software Metrics Symposium*, pp. 84–88, Nov 1998.

[33] M. M. Lehman and J. F. Ramil, "*The impact of feedback in the global software process*". *J. Syst. & Softw.* **46(2-3)**, pp. 123–134, Apr 1999.

[34] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perrry, and W. M. Turski, "*Metrics and laws of software evolution – the nineties view*". In 4th *Intl. Software Metrics Symp.*, pp. 20–32, Nov 1997.

[35] T. McCabe, "*A complexity measure*". *IEEE Trans. Softw. Eng.* **2(4)**, pp. 308–320, 1976.

[36] T. Mens, J. Fernández-Ramil, and S. Degrandsart, "*The evolution of Eclipse*". In *Intl. Conf. Softw. Maintenance*, pp. 386–395, Sep 2008.

[37] E. Mills, *Software Metrics*. Tech. Rep. Curriculum Module SEI-CM-12-1.1, Software Engineering Institute, December 1988.

[38] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "*Two case studies of open source software development: Apache and Mozilla*". *ACM Trans. Softw. Eng. & Methodology* **11(3)**, pp. 309–346, Jul 2002.

[39] MSDN, "*Visual Studio 2008: Avoid excessive complexity*". URL http://msdn.microsoft.com/en-us/library/ms182212.aspx, 2008.

[40] G. Myers, "*An extension to the cyclomatic measure of program complexity*". *SIGPLAN Notices* **12(10)**, pp. 61–64, Oct 1977.

[41] A. J. Offutt, M. J. Harrold, and P. Kolte, "*A software metric system for module coupling*". *J. Syst. & Softw.* **20(3)**, pp. 295–308, Mar 1993.

[42] P. Oman and J. Hagemeister, "*Construction and testing of polynomials predicting software maintainability*". *J. Syst. & Softw.* **24(3)**, pp. 251–266, Mar 1994.

[43] D. L. Parnas, "*Software aging*". In 16th *Intl. Conf. Softw. Eng.*, pp. 279–287, May 1994.

[44] J. W. Paulson, G. Succi, and A. Eberlein, "*An empirical study of open-source and closed-source software products*". *IEEE Trans. Softw. Eng.* **30(4)**, pp. 246–256, Apr 2004.

[45] V. T. Rajlich and K. H. Bennett, "*A staged model for the software life cycle*". *Computer* **33(7)**, pp. 66–71, Jul 2000.

[46] S. Ratanotayanon and S. E. Sim, "*Inventive tool use to comprehend big code*". *IEEE Softw.* **25(5)**, pp. 91–92, Sep/Oct 2008.

[47] E. S. Raymond, "*The cathedral and the bazaar*". URL http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar, 2000.

[48] G. Robles, J. J. Amor, J. M. Gonzalez-Barahona, and I. Herraiz, "*Evolution and growth in large libre software projects*". In 8th *Intl. Workshop Principles of Software Evolution*, pp. 165–174, Sep 2005.

[49] D. A. Rusling, "*The Linux kernel*". URL http://tldp.org/LDP/tlk/.

[50] W. Scacchi, "*Understanding open source software evolution*". In *Software Evolution and Feedback: Theory and Practice*, N. H. Madhavji, J. Fernández-Ramil, and D. E. Perry (eds.), chap. 9, pp. 181–205, Wiley, 2006.

[51] S. R. Schach, B. Jin, D. R. Wright, G. Z. Heller, and A. J. Offutt, "*Maintainability of the Linux kernel*". *IEE Proc.-Softw.* **149(2)**, pp. 18–23, 2002.

[52] S. R. Schach, B. Jin, L. Yu, G. Z. Heller, and J. Offutt, "*Determining the distribution of maintenance categories: Survey versus measurement*". *Empirical Softw. Eng.* **8**, pp. 351–365, 2003.

[53] M. Shepperd, "*A critique of cyclomatic complexity as a software metric*". *Software Engineering J.* **3**, pp. 30–36, Mar 1988.

[54] M. Shepperd and D. C. Ince, "*A critique of three metrics*". *J. Syst. & Softw.* **26**, pp. 197–210, Sep 1994.

[55] N. Smith, A. Capiluppi, and J. F. Ramil, "*A study of open source software evolution data using qualitative simulation*". *Softw. Process Improvement & Pract.* **10(3)**, pp. 287–300, Jul/Sep 2005.

[56] D. Spinellis, "*A tale of four kernels*". In 30th *Intl. Conf. Softw. Eng.*, pp. 381–390, May 2008.

[57] SRI, "*Software technology roadmap: Cyclomatic complexity*". In URL http://www.sei.cmu.edu/str/str.pdf, 1997.

[58] L. Thomas, *An Analysis of Software Quality and Maintainability Metrics with an Application to a Longitudinal Study of the Linux Kernel*. Ph.D. thesis, Vanderbilt University, 2008.

[59] L. Torvalds, "*The Linux edge*". *Comm. ACM* **42(2)**, pp. 38–39, Apr 1999.

[60] W. M. Turski, "*Reference model for smooth growth of software systems*". *IEEE Trans. Softw. Eng.* **22(8)**, pp. 599–600, Aug 1996.

[61] E. VanDoren, *Maintainability Index Technique for Measuring Program Maintainability*. Tech. rep., Software Engineering Institute, Mar 2002.

[62] E. J. Weyuker, "*Evaluating software complexity measures*". *IEEE Trans. Softw. Eng.* **14(9)**, pp. 1357–1365, Sep 1988.

[63] L. Yu, "*Indirectly predicting the maintenance effort of open-source software*". *J. Softw. Maintenance & Evolution: Res. & Pract.* **18(5)**, pp. 311–332, Sep/Oct 2006.

[64] L. Yu, S. R. Schach, K. Chen, and J. Offutt, "*Categorization of common coupling and its application to the maintainability of the Linux kernel*". *IEEE Trans. Softw. Eng.* **30(10)**, pp. 694–706, Oct 2004.